# Embedded Linux Quick Start Guide

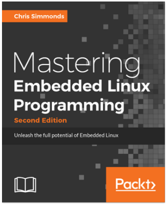
## using Buildroot and BeagleBone Black

2net Ltd

v1.5, April 2019

# About Chris Simmonds




- Consultant and trainer
- Author of *Mastering Embedded Linux Programming*
- Working with embedded Linux since 1999
- Android since 2009
- Speaker at many conferences and workshops

"Looking after the Inner Penguin" blog at `http://2net.co.uk/`

@2net_software

`https://uk.linkedin.com/in/chrisdsimmonds/`

# Overview

- This is a quick introduction to embedded Linux
- It includes hands-on exercises that will take you from zero to command prompt in one day
- The target board is a BeagleBone Black
- It uses Buildroot to generate the Linux distro

# Topics

- Getting started with embedded Linux
- Build systems
- The toolchain
- Device trees
- Accessing hardware

# Introduction to embedded Linux

# What is embedded Linux?

- Embedded Linux = Linux running on embedded hardware
  - "Linux" in the broad sense: a Linux kernel plus other open source packages needed to make a working system
- Hard to define: applications range from the small (light bulbs) to the large (industrial plant)
- ... from the trivial (light bulbs) to the critical (industrial plant)
  - Linux is commonly used in mission critical applications, but not (yet) safely critical

# Minimum hardware spec

- 32 or 64-bit processor architecture
  - examples: ARM, PPC, MIPS, SH, x86
- At least 16 MiB RAM (*)
- At least 4 MiB storage (*), usually flash memory
- Also uClinux (www.uclinux.org) for processors without memory management unit
  - Examples: ADI Blackfin, Altera NIOS, Xilinx MicroBlaze, ARM Cortex M-3

(*) It is possible to build Linux systems with less RAM and flash, but it requires non-trivial effort

# Driving factors

- **Moore's law**: complex hardware requires complex software
- **Free**: you have the **freedom** to get and modify the code, making it easy to adapt and extend
- **Functional**: supports a (very) wide range of hardware
- **Up to date**: the kernel has a three month release cycle
- **Free**: there is no charge for using the source code

# Pain points

- Lack of support for your particular hardware (always check with the manufacturer before you design a component in)
- The rapid update cycle does not fit well with the slower cycle for embedded projects
- SoC/SoM/SBC vendors do not always push fixes and features as quickly as we would like
- Lack of knowledge (that’s why I am here)

# Working with open source licenses

- All software used on this course is open source
    - You have the *freedom* to modify and redistribute the source code
- Various open source licenses, but the main ones are
    - "permissive", such as BSD, MIT and Apache
    - "copyleft" - the GPL (General Public License)
- The license should be part of each package
- In a file named `LICENSE` or `COPYING`, and is usually at the beginning of each source file

# Permissive licenses

- In general, these licenses state that you can create derivative works so long as you
  - Don't change copyright notices
  - Don't change the limited warranty notice
- You don't need to distribute source code

I am not a lawyer. Please consult your legal department for clarification

# GPL v2

- Version 2 of the General Public License says
  - You can create derivative works
  - You must distribute source code to end users
    - by public server
    - or by "written offer": a promise to supply code on request
  - You are creating a derivative work if you link with code or a library licensed under GPL

Note: I am not a lawyer. Please consult your legal department for clarification

# LGPL

- The lesser GPL (LGPL) license is mostly applied to library code
- Allows linking to a library without creating a derivative work
  - i.e. you can write proprietary programs that link dynamically with LGPL libraries
  - Static linking is a more complex legal issue: don’t do it

Note: I am not a lawyer. Please consult your legal department for clarification

# GPLv3 and LGPLv3

- Adds "The right to tinker"
  - it must be possible to replace the GPLv3 components of any device
  - also known as the "anti Tivoization clause"
- and protection against patent threats
  - You must provide every recipient with any patent licenses necessary to exercise the rights that the GPLv3 gives them
- and many other details...

Note: I am not a lawyer. Please consult your legal department for clarification

# Open source: good or bad?

- Overwhelmingly good!
  - Much wider code review leading to higher quality
  - Easy to share code with others
  - Gives you access to a very big pool of code
  - The individual programmer gets recognition

# Elements of embedded Linux

Every embedded Linux project has these four elements:

- Toolchain: to compile all the other elements
- Bootloader: to initialise the board and load the kernel
- Kernel: to manage system resources
- Root filesystem: to run applications

# Element 1: Toolchain

- Toolchain = GNU GCC + C library + GNU GDB
  - LLVM/Clang is also an option, but not quite mainstream yet
- Native toolchain
  - Install and develop on the target
- Cross toolchain
  - Build on development system, deploy on target
  - Keeps target and development environments separate
- Cross toolchains are the most common

# Element 2: Bootloader

- Tasks
  - Initialise the board
  - Load the kernel
  - Maintenance tasks, e.g. flash system images
- Open source bootloaders
  - Das U-Boot
  - Little Kernel
  - GRUB 2 (for X86 and X86_64)

# Das U-Boot

`www.denx.de/wiki/U-Boot`

- Open source, GPL license
- Small run-time binary (50 - 800 KiB)
- Supports many CPU architectures, incl. ARM, MIPS, PowerPC, SH
- ... and many boards (> 1000)
- Reasonably easy to port to a new board

# U-Boot command-line

Load a kernel image into memory from...

- NAND flash

```
=> nand read 80100000 1000000 200000
```

- eMMC or SD card

```
=> mmc rescan 1
=> fatload mmc 1:1 80100000 zimage
```

- TFTP server

```
=> setenv ipaddr 192.168.1.2
=> setenv serverip 192.168.1.1
=> tftp 80100000 zImage
```

- Boot a kernel image (already loaded into memory)

```
=> bootz 80100000
```

# U-Boot environment

- Contains parameters, e.g. `ipaddr` and `serverip` from previous slide
- Parameters can be created or modified at run-time using `setenv`
- Default set is hard-coded in U-Boot
- At boot, additional parameters may be read from:
  - Dedicated area of flash memory
  - A file named `boot.scr`
  - A file named `uEnv.txt`

# Element 3: Kernel

`https://www.kernel.org`

- Tasks
  - Manage system resources: CPU, memory, I/O
  - Interface with hardware via device drivers
  - Provide a (mostly) hardware-independent API
- Rapid development cycle: new version every 12 weeks

# Kernel versions

Up to May 2011 2.6.39.1

Release number: changes every 12 weeks or so

Bug fix number: changes every time a bug is fixed

From July 2011 3.0.1

From April 2015 4.0.1

From January 2019 5.0.1

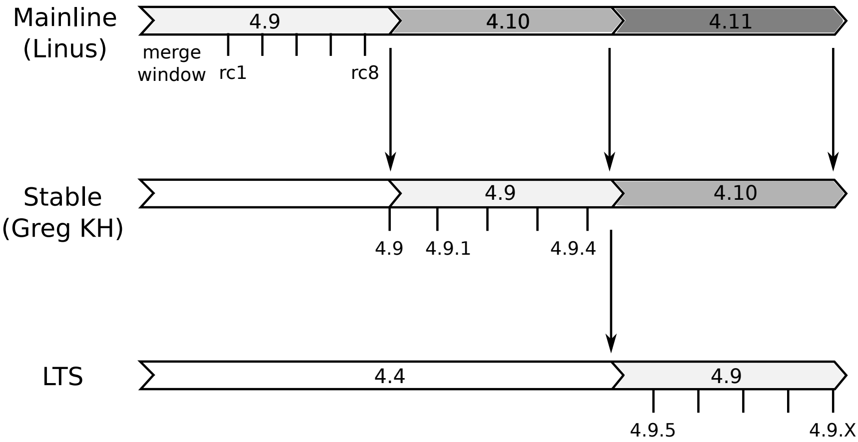
# Mainline, stable and LTS

- **Mainline:** Linus Torvald's development tree
- **Stable:** bug fixes to the previous mainline release
- **LTS** (Long Term Support): versions supported for >= 2 years
- Current LTS kernels:

| Version | Maintainer | Released | EOL |
|---|---|---|---|
| 4.19 | Greg KH | 2018-10-22 | Dec, 2020 |
| 4.14 | Greg KH | 2017-11-12 | Jan, 2020 |
| 4.9 | Greg KH | 2016-12-11 | Jan, 2023 |
| 4.4 | Greg KH | 2016-01-10 | Feb, 2022 |
| 3.16 | Ben Hutchings | 2014-08-03 | Apr, 2020 |

Reference:
https://www.kernel.org/category/releases.html

# Timeline


Mainline
(Linus)
4.9
4.10
4.11
merge
window
rc1
rc8
Stable
(Greg KH)
4.9
4.10
4.9
4.9.1
4.9.4
LTS
4.4
4.9
4.9.5
4.9.X

# Vendor kernels

- Mainline Linux has good support for x86/x86_64
- SoC vendors take mainline Linux and customise to support their chips
  - Vendor kernels lag behind mainline
  - Vendors don't take every mainline release: typically only one per year
  - Most vendors are not very good at pushing their changes into mainline
- *Most of the time you will be not be working with mainline Linux*

# Embedded build systems

- Building the four elements by hand is time consuming
- Embedded build systems make it easy

| Tool | Notes |
|---|---|
| buildroot | Small, menu-driven |
| OpenWrt | A variant of Buildroot for network devices |
| OpenEmbedded | General purpose |
| Yocto Project | General purpose, wide industry support, complex |

# Summary

- Embedded Linux is just Linux used in an embedded environment
- The basis of embedded Linux is
    - Toolchain, bootloader, kernel and root filesystem
- Build systems take the hard work out of embedded Linux

# Build systems

# Overview

- Embedded build systems
- Buildroot
  - Packages
  - Board configuration files

# Build systems

- Automate production of embedded Linux
- Build from up-stream source some or all of
  - Toolchain
  - Bootloader
  - Kernel
  - Root filesystem

# Buildroot

- One of the first embedded build systems (2001)
  - (OpenEmbedded started two years later)
- Web: `https://buildroot.org`
- As well as the **root filesystem**, can also build **toolchain**, **bootloader**, and **kernel**
- Architectures: ARM, PowerPC, MIPS, X86, and many more...
- Packages: over 1500
- Board configs: over 150

# Configuration

- Uses the same Kbuild build system as the kernel
- Run-time configuration information is stored in `.config`
- Default configuration files for many boards in `configs/`
- Edit configuration using menuconfig (also xconfig and gconfig)

# Outputs

- Downloaded source code -> `dl/`
- Build artifacts -> `output/`
- `output/` contains

| Directory | Notes |
|---|---|
| build | Working directory for compiling source |
| host | Tools that run on the host, incl. toolchain |
| images | bootloader, kernel and root filesystem for the target |
| staging | Link to the sysroot of the toolchain |
| target | Staging area for target root filesystem |

# Packages

- **Packages** are programs and libraries for the target or host
- Each has a subdirectory in `package/`
- ... which contains
  - `Config.in`: configuration points that can be selected by the menu editor
  - `[package].mk`: a GNU make fragment that builds the package
  - `[package].hash`: (optional) a hash of the source archive, used to detect corrupt downloads

# Package Config.in

- In the same format as kernel Kconfig files
- Example `package/tree/Config.in`:

```
config BR2_PACKAGE_TREE
        bool "tree"
        depends on BR2_USE_WCHAR
        help
          Tree is a recursive directory listing command that produces
          a depth indented listing of files, which is colorized ala
          dircolors if the LS_COLORS environment variable is set and
          output is to tty.

          http://mama.indstate.edu/users/ice/tree/

comment "tree needs a toolchain w/ wchar"
        depends on !BR2_USE_WCHAR
```

# Package make file

- Example: `package/tree/tree.mk`

```
[...]
TREE_VERSION = 1.7.0
TREE_SOURCE = tree-$(TREE_VERSION).tgz
TREE_SITE = http://mama.indstate.edu/users/ice/tree/src
TREE_LICENSE = GPL-2.0+
TREE_LICENSE_FILES = LICENSE

define TREE_BUILD_CMDS
        $(MAKE) $(TARGET_CONFIGURE_OPTS) -C $(@D)
endef

define TREE_INSTALL_TARGET_CMDS
        $(INSTALL) -D -m 0755 $(@D)/tree $(TARGET_DIR)/usr/bin/tree
endef

$(eval $(generic-package))
```

# Package hash file

- Format: `[type] [hash] [file name]`
- `[type]` can be one of md5, sha1, sha224, sha256, sha384, sha512, none
  - Use `none` for code obtained from a repository (git, subversion, ...)
- Example: `package/tree/tree.hash`

```
# Locally calculated
sha256  6957c20e82561ac4231638996e74f4cfa4e6faabc5a2f511f0b4e3940e8f
  tree-1.7.0.tgz
```

# Boards

- Board-specific configuration goes in directory `board/`
- Directory name convention:
  `board/[manufacturer]/[board]`
- Recommended layout within [board] directory:

```
patches/            - Patches
post-build.sh       - Script run after build but before the
                      image has been created
post-image.sh       - Script run after image has been created
readme.txt          - Description of this BSP
rootfs_overlay/     - Files copied into rootfs
```

# Overlays

- Simple method of adding your own files to the rootfs images
- The contents of the overlay directory are copied over the rootfs before creating the images
- Recommended name is `rootfs_overlay/`
- Set the path of the overlay directory in `BR2_ROOTFS_OVERLAY` (in the **System configuration** menu)

# Configuration

- Keep a copy of board configuration for others
- Many are stored in directory `configs/`
- Use `make savedefconfig` to create a small config file which records only the changes from the default

```
$ make savedefconfig
$ cp defconfig configs/[boardname]_defconfig
```

# OpenEmbedded

- `www.openembedded.org`
- Based on recipes grouped together into **meta layers**
- The recipes are processed by a task scheduler named **BitBake**
- Recipes generate packages as RPM (default)
- In other words, OpenEmbedded is a tool to create a custom Linux distribution

# OpenEmbedded Core

- The core of OpenEmbedded, **oe core**, is the basis of several build systems
  - OpenEmbedded itself
  - Poky (part of the Yocto Project)
  - ELDK (from Denx)
  - Mentor Graphics Linux
  - ... and others

# The Yocto Project

- The Yocto Project is a Linux Foundation project to maintain a build system for embedded Linux
- Consists of
  - oe-core, shared with OpenEmbedded
  - BitBake: shared with OpenEmbedded
  - Poky, the distribution metadata
  - Reference BSPs including BeagleBone
  - Documentation, which is extensive
  - Toaster: a graphical user interface for Yocto

# Buildroot vs Yocto Project

Buildroot is good because:

- Very easy to set up and use
- Builds are fast (about half an hour)
- Good for demos; one of a kind projects; small teams

Yocto Project/OpenEmbedded is good because:

- Distro/Machine/Image trio encourage re-use of recipes
- Layers partition recipes and make it easy to import recipes from others
- Wide industry support
- Good for large, distributed teams; products with many variants

# Debian

- Embedded != data centre
- Challenges using a full Linux distribution
  - need to slim down to reduce flash memory usage and reduce attack surface
  - need to reduce number of log writes (logs generate a large number of short writes which are bad news for flash memory)
  - Need to turn off swap (also bad for flash)
  - apt/zypper updates are not atomic and may corrupt the system if interrupted
- Debian is good for quick demos/PoC
- More often used on embedded PC (x86) hardware

# Summary

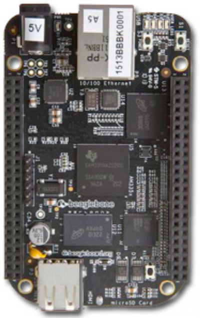
- Buildroot is a powerful build system
- Ideal for demonstrations and quick builds
- However, it does not scale as well at Yocto Project/Open Embedded

# The BeagleBone Black

# The BeagleBone Black

- Open source hardware design from `http://beagleboard.org`
- Low cost ($55)
- Extensible via stackable daughter boards, called *capes*

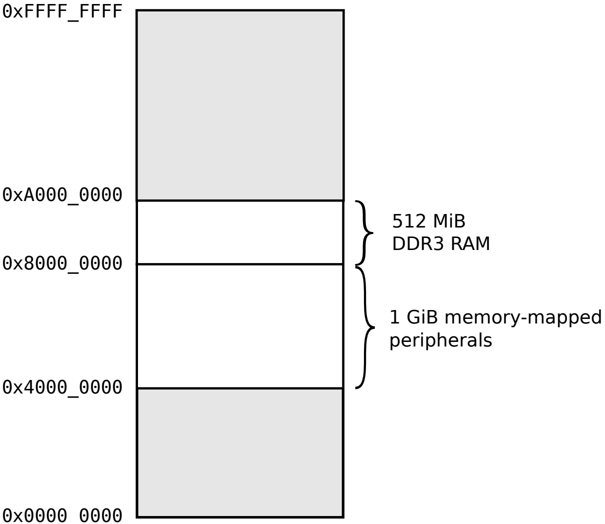


# Features

- TI AM335x 1GHz Cortex-A8 SoC
- Imagination Tech. PowerVR SGX530 GPU
- 512 MiB DDR3 RAM
- 2 or 4 GiB 8-bit eMMC on-board flash storage
- MicroSD card slot for external storage
- Mini USB OTG port, also provides power
- Full size USB 2.0 host
- 10/100 Ethernet
- Mini HDMI connector

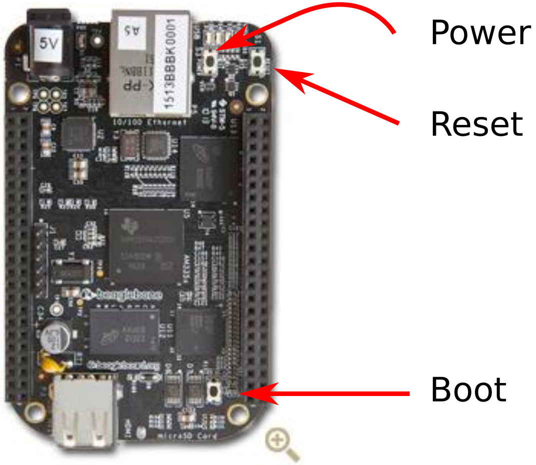
# Memory map


0xFFFF_FFFF
0xA000_0000
0x8000_0000
0x4000_0000
0x0000_0000
512 MiB
DDR3 RAM
1 GiB memory-mapped
peripherals

# Storage

- MMC0: External microSD (4-bit)
- MMC1: 2/4 GiB internal eMMC (8-bit)

# Switches



# Boot sequence

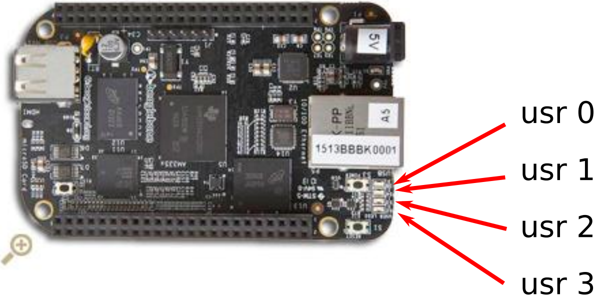
- Default:
    - boot from internal eMMC
- If **boot switch** is pressed while power is applied:
    - Try to boot from microSD card
    - If no SD card present, try to load an image via USB port, followed by serial port

# Boot files

When booting from MMC (eMMC or SD):

- The first partition is mounted
  - Must be FAT32 (vfat) format
  - Must have boot flag set
- Load and execute SPL in `MLO`
- Load and execute boot loader in `u-boot.img`
- U-Boot loads
  - Linux kernel: `zImage`
  - Device tree binary: `am335x-boneblack.dtb`
  - (Optinally) an intial RAM disk
- U-Boot starts Linux

# LEDs



# Toolchain

# Overview

- Types of toolchain
- Cross compiling
- Reference: MELP2 chapter 2

# The toolchain

- Toolchain = GNU GCC + C library + GNU Binutils + GNU GDB
  - LLVM/Clang is also an option, but not quite mainstream yet
- Native toolchain
  - Install and develop on the target
- Cross toolchain
  - Build on development system, deploy on target
  - Keeps target and development environments separate
- Cross toolchains are the most common

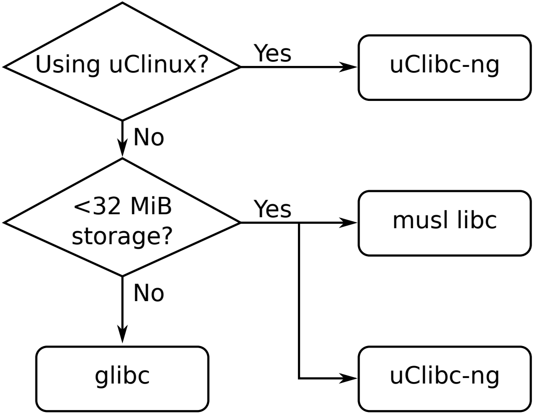
# Choosing a C library (1)

- The C library is the interface between user space and kernel
- Three options:

| Library | License | Notes |
|---|---|---|
| GNU glibc | LGPL2.1 | Big, good support for POSIX and extensions https://www.gnu.org/software/libc |
| musl libc | MIT | Small: a good choice for memory-constrained systems http://www.musl-libc.org |
| uClibc NG | LGPL2.1 | Small, well established but development seems to be less active than musl http://www.uclibc-ng.org |

# Choosing a C library (2)



# Getting a toolchain

Your options are:

- Build from upstream source, e.g. using CrosstoolNG: `http://crosstool-ng.github.io`
- Download from a trusted third party, e.g. Linaro or Bootlin
- Use the one provided by your SoC/board vendor (check quality first)
- **Use an embedded build system (Yocto Project, OpenEmbedded, Buildroot) to generate one**

# Toolchain prefix

- GNU toolchains are usually identified by a prefix
  `arch-vendor-kernel-operating system`
- Example: `mipsel-unknown-linux-gnu-`
  - **arch**: mipsel (MIPS little endian)
  - **vendor**: unknown
  - **kernel**: linux
  - **operating system**: gnu

# Toolchain prefix for ARM toolchains

- 32-bit ARM has several incompatible ABIs
- Reflected in the **Operating system** part of the prefix
- Examples:
  - `arm-unknown-linux-gnu-`: Old ABI (obsolete)
  - `arm-unknown-linux-gnueabi-`: Extended ABI with soft floating point(*)
  - `arm-unknown-linux-gnueabihf-`: Extended ABI with hard floating point(*)

(*) Indicates how floating point arguments are passed: either in integer registers or hardware floating point registers

# sysroot

- The **sysroot** is the directory containing the supporting files
  - Include files; shared and static libraries, etc.
- Native toolchain: sysroot = '/'
- Cross toolchain: sysroot is inside the toolchain directory
- Find it using `-print-sysroot`
- Example:

```
$ aarch64-buildroot-linux-gnu-gcc -print-sysroot
/home/traning/aarch64--glibc--stable/bin/../
aarch64-buildroot-linux-gnu/sysroot
```

# sysroot

sysroot = aarch64-buildroot-linux-gnu/sysroot

```
[sysroot]
├── lib
├── lib64 -> lib
└── usr
    ├── include
    ├── lib
    │   ├── audit
    │   └── gconv
    ├── lib64 -> lib
    ├── libexec
    │   └── getconf
    └── share
        ├── aclocal
        ├── buildroot
        ├── i18n
        └── locale
```

# Getting to know your toolchain

Find out about GCC with these options

- `-print-sysroot`: print sysroot
- `--version`: version
- `-v`: configuration, look out for
  - `--enable-languages=` (example c,c++)
  - `--with-cpu=` (the default CPU)
  - `--enable-threads` (has POSIX threads library)

# The tools

| Tool | Description |
|---|---|
| addr2line | Converts program addresses into filen and line no. |
| ar | archive utility is used to create static libraries |
| as | GNU assembler |
| cpp | C preprocessor, expands #define, #include etc |
| g++ | C++ frontend, (assumes source is C++ code) |
| gcc | C frontend, (assumes source is C code) |
| gcov | code coverage tool |
| gdb | GNU debugger |
| gprof | program profiling tool |
| ld | GNU linker |
| nm | lists symbols from object files |
| objcopy | copy and translate object files |
| objdump | display information from object files |
| readelf | displays information about files in ELF object format |
| size | lists section sizes and the total size |
| strings | displays strings of printable characters in files |
| strip | strip object file of debug symbol tables |

# Cross compiling

Compile a small program:

```
$ aarch64-buildroot-linux-gnu-gcc hello-arm.c -o hello-arm
$ ls -l
total 12
-rwxrwxr-x 1 traning traning 7360 Oct 13 10:45 hello-arm
-rw-rw-r-- 1 traning traning  119 Oct 13 10:44 hello-arm.c
```

Check that it really is cross-compiled:

```
$ file hello-arm
hello-arm: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux-aarch64.so.1,
for GNU/Linux 3.10.0, not stripped
```

See the run-time linker and libraries:

```
$ ~/embedded/list-libs hello-arm
      [Requesting program interpreter: /lib/ld-linux-aarch64.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

# Summary

- The toolchain generates compiled code for the target
- We almost always use cross toolchains for embedded development

# Device trees

# Overview

- Why do we need device trees?
- Device tree syntax
- Compiling
- Reference: MELP2 chapter 3: Introducing device trees

# Why do we need device trees?

- The kernel needs to know details about hardware
  - to decide which drivers to initialise
  - to configure device parameters such as register addresses and IRQ
- Sources of information:
  - firmware ACPI tables (x86 and ARM server)
  - bus enumeration, e.g. PCI
  - hard coded structures
  - **device tree** (ARM, PPC, MIPS, and others)

# Device Tree

- Open Firmware specification, IEEE-1275-1994
- Description of hardware (contains no code)
- Tree of nodes, containing attributes, for example:

```
/dts-v1/;
/{
    model = "TI AM335x BeagleBone";
    compatible = ti,am335x-bone, "ti,am33xx";
    #address-cells = <1>;
    #size-cells = <1>;
    memory@0x80000000 {
        device_type = "memory";
        reg = <0x80000000 0x20000000>; /* 512 MB */
    };
   [...]
};
```

# Device tree binaries

- **.dts** source files are in `arch/<ARCH>/boot/dts`
- compiled to **.dtb**, using `dtc`
- dtb file is loaded into memory by the bootloader
- The U-boot `bootz` command takes three arguments

```
bootz <kernel> <ramdisk> <dt binary>
```

- If there is no ramdisk (which is common)

```
bootz <kernel> - <dt binary>
```

- The dtb is also known as a **Device Tree Blob**, or a **Flattened Device Tree** (fdt)

# Basic structure

- Represents hardware as a hierarchy
- Starts at a root node, named "/"
- Nodes may contain child nodes
- Each node contains name = value pairs
- Must contain version: `/dts-v1/;`
- Comments are C style: `/* this is a comment */`

# Node names

- Node names are up to 31 characters long
- May include an '@' sign and an address, e.g. if there is more than one

```
/ {
    compatible = "ti,omap4430", "ti,omap4";
    cpus {
        cpu@0 {
            compatible = "arm,cortex-a9";
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
        };
    };
};
```

# The compatible property

- Every node that represents a device has a **compatible** property
- The kernel uses **compatible** to choose which driver to use
- To avoid name collisions, often it is of the form `"<manufacturer>,<model>"`

```
wdt2: wdt@44e35000 {
    compatible = "ti,omap3-wdt";
[...]
```

- There may be more than one entry
- Starts with most compatible (exact match)

```
serial@44e09000 {
        compatible = "ti,am3352-uart", "ti,omap3-uart";
[...]
```

# Top-level compatible property 1/2

arch/arm/boot/dts/am335x-boneblack.dts

```
/ {
    model = "TI AM335x BeagleBone Black";
    compatible = "ti,am335x-bone-black", "ti,am335x-bone", "ti,am33xx";
};
```

- Matches a `DT_MACHINE_START`, starting with the one on the left
- For BBB, first match is on `ti,am33xx`, shown on next slide
- Installs function pointers for `init_machine`, etc

# Top-level compatible property 2/2

arch/arm/mach-omap2/board-generic.c

```
#ifdef CONFIG_SOC_AM33XX
static const char *const am33xx_boards_compat[] __initconst = {
        "ti,am33xx",
        NULL,
};

DT_MACHINE_START(AM33XX_DT, "Generic AM33XX (Flattened Device Tree)")
        .reserve        = omap_reserve,
        .map_io         = am33xx_map_io,
        .init_early     = am33xx_init_early,
        .init_machine   = omap_generic_init,
        .init_late      = am33xx_init_late,
        .init_time      = omap3_gptimer_timer_init,
        .dt_compat      = am33xx_boards_compat,
        .restart        = am33xx_restart,
MACHINE_END
#endif
```

# The reg property

- Device addresses are given by the **reg** property
- **reg** is an array of cell values
- A cell is a 32-bit value

```
#address-cells = <1>;
#size-cells = <1>;
memory@0x80000000 {
    device_type = "memory";
    reg = <0x80000000 0x20000000>; /* 512 MB */
};
```

- A reg can contain one or more pairs of base and size
- `reg = <base1 size1 [base2 size2 [...]]>;`

# address-cells and size-cells

- Addresses can be of different lengths so don't always fit a single 32-bit cell
- The number of cells for base and size are given in the **parent node**
  - `#address-cells` - number of cells for base
  - `#size-cells` - number of cells for size

# Example 1

- The cpu property has an address that is a simple number
- `#address-cells` is 1, `#size-cells` is 0

```
cpus {
    #address-cells = <1>;
    #size-cells = <0>;
    cpu@0 {
        device_type = "cpu";
        compatible = "arm,cortex-a15";
        reg = <0>;
    };
    cpu@1 {
        device_type = "cpu";
        compatible = "arm,cortex-a15";
        reg = <1>;
};
```

# Example 2

- On a device with 64-bit addressing, you need two cells for each address

```
/ {
    #address-cells = <2>;
    #size-cells = <2>;
    memory@80000000 {
        device_type = "memory";
        /*  2GB @ 0x80000000 */
        reg = <0x00000000 0x80000000 0x0 0x40000000>;
    };
}
```

# Labels and Phandles

- The device tree hierarchy represents bus connections
- Some things cut across that structure
  - Interrupts, clocks, power
- A phandle is a label for a node that can be referenced elsewhere

```
intc: interrupt-controller@48200000 {
    ...
};
```

# Phandle example

```
{
    intc: interrupt-controller@48200000 {
        compatible = "ti,am33xx-intc";
        interrupt-controller;
        #interrupt-cells = <1>;
        reg = <0x48200000 0x1000>;
    };
        serial@44e09000 {
        compatible = "ti,omap3-uart";
        ti,hwmods = "uart1";
        clock-frequency = <48000000>;
        reg = <0x44e09000 0x2000>;
        interrupt-parent = <&intc>;
        interrupts = <72>;
    };
}
```

# Where is the phandle?

- dtc creates a phandle from a label when it sees a reference from another node
- Decompiling the dtb shows the actual code (next slide)

# Where is the phandle?

```
{
    interrupt-controller@48200000 {
        compatible = "ti,am33xx-intc";
        interrupt-controller;
        #interrupt-cells = <0x1>;
        reg = <0x48200000 0x1000>;
        linux,phandle = <0x1>;
        phandle = <0x1>;              <-------- declaration
        };

    serial@44e09000 {
        compatible = "ti,omap3-uart";
        ti,hwmods = "uart1";
        clock-frequency = <0x2dc6c00>;
        reg = <0x44e09000 0x2000>;
        interrupts = <0x48>;
        status = "okay";
        interrupt-parent = <0x1>;      <------- reference
    };
```

# Interrupts 1/2

- An interrupt controller has properties:
  - `interrupt-controller;`
  - `#interrupt-cells = <n>` where **n** is the number of cells for an **interrupt specifier**
- The format of an interrupt specifier is device-dependent
- Often, it is just one number: the IRQ
  - this is the case with the AM335x on the BBB

# Interrupts 2/2

- A device that generates interrupts has properties:
  - `interrupt-parent`: defines which interrupt controller it is connected to (*)
  - `interrupts`: defines interrupt specifiers(s)

(*) May inherit from parent node

# Include files

- Where boards or SoCs share device definitions common code is placed in **.dtsi** include files
- There are two conventions

1: the Open Firmware standard

```
/include/ "vexpress-v2m.dtsi"
```

2: C style includes

```
#include "am33xx.dtsi"
#include <dt-bindings/pinctrl/am33xx.h>
```

- For case (2) the dts is passed through the C preprocessor, cpp, to resolve the `#include` and `#define` macros

# Modifying a node 1/3

- You can refer to the same node multiple times
  - Properties are combined, later ones overwrite earlier ones
- Nodes may be referenced by phandle or full path

# Modifying a node 2/3

## Example 1. - reference via phandle

`am33xx.dtsi` has uarts 0..6 with status disabled

```
uart0: serial@44e09000 {
    compatible = "ti,omap3-uart";
    [...]
    status = "disabled";
};
```

In a board-specific file, `am335x-bone-common.dtsi`, `uart0` is enabled using its phandle as a reference:

```
&uart0 {
    status = "okay";
};
```

# Modifying a node 3/3

Example 2. reference by full path, starting at the root

The Beaglebone Black has a HDMI interface which the other BeagleBones do not have. It it is declared in `am335x-boneblack.dts` like this:

```
/ {
    hdmi {
        compatible = "ti,tilcdc,slave";
        [...]
        status = "okay";
    };
};
```

# Standard bindings

- Most device types have additional properties
- The format must match what the kernel is expecting
- Standard bindings are defined in `$LINUXSRC/Documentation/devicetree/bindings` (since Linux 3.0)

# Compiling device trees

- There is a copy of dtc in the Linux source, in `scripts/dtc/dtc`
- To compile

```
$ scripts/dtc/dtc -O dtb -o fdt-with-LCD4.dtb -I dts fdt-with-LCD4.dts
```

- To de-compile

```
$ scripts/dtc/dtc -O dts -o fdt-with-LCD4.dts -I dtb fdt-with-LCD4.dtb
```

# Device tree at run-time

- The device tree can be read from `/proc/device-tree` or `/sys/firmware/devicetree/base`
- The data is in binary format, as stored in the .dtb file

```
# hexdump -C /sys/firmware/devicetree/base/memory/reg
00000000  80 00 00 00 20 00 00 00                           |.... ...|
```

The address is 0x80000000 and the size is 0x20000000

There is also a complete copy of the dt binary in `/sys/firmware/fdt`

# Summary

- Device trees contain a description of the hardware
- They separate out the details from the device driver code
- The represent hardware as a hierarchy, which roughly corresponds to the hardware bus structure
- You can refer to another node using a phandle

# Accessing hardware from userspace

# Overview

- Generic kernel device drivers
- GPIO
- I2C

# Accessing kernel drivers

- In Linux, everything is a file [1]
- Applications interact with drivers via POSIX functions open(2), read(2), write(2), ioctl(2), etc
- Two main types of interface:

1. Device nodes in `/dev`
   - For example, the serial driver, `ttyS`. Device nodes are named `/dev/ttyS0`, `/dev/ttyS1` ...
2. Driver attributes, exported via *sysfs*
   - For example `/sys/class/gpio`

[1] Except network interfaces, which are sockets

# Userspace drivers

- **Userspace drivers** keep most of the logic in userspace and use **generic kernel drivers** to access the hardware
- We will look at:
  - GPIO
  - I2C

# A note about device trees

- Even though you are writing userspace drivers, you still need to make sure that the hardware is accessible to the kernel
- On ARM based systems, this may mean changing the device tree or adding a device tree overlay

# GPIO: General Purpose Input/Output

- Pins that can be configured as inputs or outputs
- As outputs:
  - used to control LEDs, relays, control chip selects, etc.
- As inputs:
  - used to read a switch or button state, etc.
  - some GPIO hardware can generate an interrupt when the input changes

# Two userspace drivers!

- **gpiolib**[1]: old, but scriptable interface using sysfs
- **gpio-cdev**: new, higher performance method using character device nodes `/dev/gpiochip*`

[1] it's **not** a library

# The gpiolib sysfs interface

- GPIO pins grouped into registers, named **gpiochipNN**
- Each pin is assigned a number from 0 to XXX

```
# ls /sys/class/gpio/
export   gpiochip0   gpiochip32   gpiochip64   gpiochip96   unexport
```

This device has 4 gpio chips
each with 32 pins

Write to this
file to export
a GPIO pin
to user space

Write to this
file to unexport
a GPIO pin
to user space

# Inside a gpiochip

```
# /sys/class/gpio/gpiochip0
base  device   label  ngpio  power   subsystem  uevent
```

The number of GPIO pins (32)

A lable to identify the chip
(gpiochip0)

The starting GPIO number (0)

# Exporting a GPIO pin

```
# echo 42 > /sys/class/gpio/export
# ls /sys/class/gpio
export   gpio42  gpiochip0   gpiochip32   gpiochip64   gpiochip96   unexport
```

If the export is successful, a new directory is created

# Inputs and outputs

```
# ls /sys/class/gpio/gpio42
active_low  device  direction  edge  power  subsystem  uevent  value
```

Set to 1 to invert input and ouput

Set direction by writing "out" or "in". Default "in"

The logic level of the pin. Change the level of outputs by writing "0" or "1"

# Interrupts

- If the GPIO can generate interrupts, the file **edge** can be used to control interrupt handling
- edge = ["none", "rising", "falling","both"]
- For example, to make GPIO60 interrupt on a falling edge:
  - `echo falling > /sys/class/gpio/gpio60/edge`
- To wait for an interrupt, use the poll(2) function

# The gpio-cdev interface

- One device node per GPIO register named `/dev/gpiochip*`
- Access the GPIO pins using `ioctl(2)`
- Advantages
    - Naming scheme gpiochip/pin rather than uniform but opaque name space from 0 to XXX
    - Multiple pin transitions in single function call without glitches
    - More robust handling of interrupts

# gpio-cdev example 1/2

```
/*
 * Demonstrate using gpio cdev to output a single bit
 * On a BeagleBone Black, GPIO1_21 is user LED 1
 */

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/gpio.h>

int main(void)
{
    int f;
    int ret;
    struct gpiohandle_request req;
    struct gpiohandle_data data;
```

# gpio-cdev example 2/2

```
    f = open("/dev/gpiochip1", O_RDONLY);
    req.lineoffsets[0] = 21;
    req.flags = GPIOHANDLE_REQUEST_OUTPUT; /* Request as output */
    req.default_values[0] = 0;
    strcpy(req.consumer_label, "gpio-output"); /* up to 31 characters */
    req.lines = 1;

    ret = ioctl(f, GPIO_GET_LINEHANDLE_IOCTL, &req);

    /* Note that there is a new file descriptor in req.fd to handle the
       GPIO lines */
    data.values[0] = 1;
    ret = ioctl(req.fd, GPIOHANDLE_SET_LINE_VALUES_IOCTL, &data);
    close(f);
    return 0;
}
```

# I2C: the Inter-IC bus

- Simple 2-wire serial bus, commonly used to connect sensor devices
- Each I2C device has a 7-bit address, usually hard wired
- 16 bus addresses are reserved, giving a maximum of 112 nodes per bus
- The master controller manages read/write transfers with slave nodes

# The i2c-dev driver

- **i2c-dev** exposes I2C master controllers
- Need to load/configure the i2c-dev driver (`CONFIG_I2C_CHARDEV`)
- There is one device node per i2c master controller

```
# ls -l /dev/i2c*
crw-rw---T 1 root i2c 89, 0 Jan  1  2000 /dev/i2c-0
crw-rw---T 1 root i2c 89, 1 Jan  1  2000 /dev/i2c-1
```

- You access I2C slave nodes using read(2), write(2) and ioctl(2)
- Structures defined in `usr/include/linux/i2c-dev.h`

# Detecting i2c slaves using i2cdetect

- **i2cdetect**, from i2c-tools package, lists i2c adapters and probes devices
  - Example: detect devices on bus 1 (`/dev/i2c-1`)

```
# i2cdetect -y -r 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- 39 -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- UU UU UU UU -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

UU = device already handled by kernel driver

0x39 = device discovered at address 0x39

# i2cget/i2cset

- `i2cget <bus> <chip> <register>`: read data from an I2C device
  - Example: read register 0x8a from device at 0x39

```
# i2cget -y 1 0x39 0x8a
0x50
```

- `i2cset <bus> <chip> <register>`: writedata to an I2C device
  - Example: Write 0x03 to register 0x80:

```
# i2cset -y 1 0x39 0x80 3
```

# I2C code example - light sensor, addr 0x39

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/i2c-dev.h>

int main(int argc, char **argv)
    int f;
    char buf[4];

    f = open("/dev/i2c-1", O_RDWR);
    ioctl(f, I2C_SLAVE, 0x39) < 0) {

    buf[0] = 0x8a;              /* Chip ID register */
    write(f, buf, 1);
    read(f, buf, 1);
    printf("ID 0x%x\n", buf [0]);
}
```

Code: https://github.com/csimmonds/userspace-io-ew2016

# Other examples

- SPI: access SPI devices via device nodes `/dev/spidev*`
- USB: access USB devices via libusb
- User defined I/O: UIO
    - Generic kernel driver that allows you to write userspace drivers
    - access device registers and handle interrupts from userspace

# Summary

- Using generic device drivers avoids putting logic into the kernel
    - Kernel code can be hard to debug, and may have licensing issues

# Conclusion

# The story so far

- We looked at the *four elements of embedded Linux*:
    - Toolchain
    - Bootloder
    - Kernel
    - Root filesystem
- We used **Buildroot** to create these elements
- We tried it out using a BeagleBone Black target

# Next steps

- Keep on learning
- Other classes that may interest you:
  - Mastering Yocto Project
  - Embedded Linux System Programming
  - Device Drivers for Embedded Linux